Natural Language Processing

© 李浩东 3190104890@zju.edu.cn

- Long Short-Term Memory (LSTM)
- Gated Recurrent Units (GRU)
- Deep Recurrent Neural Networks
- Bidirectional Recurrent Neural Networks
- Encoder-Decoder Seq2Seq for Machine Translation

Long Short-Term Memory (LSTM)

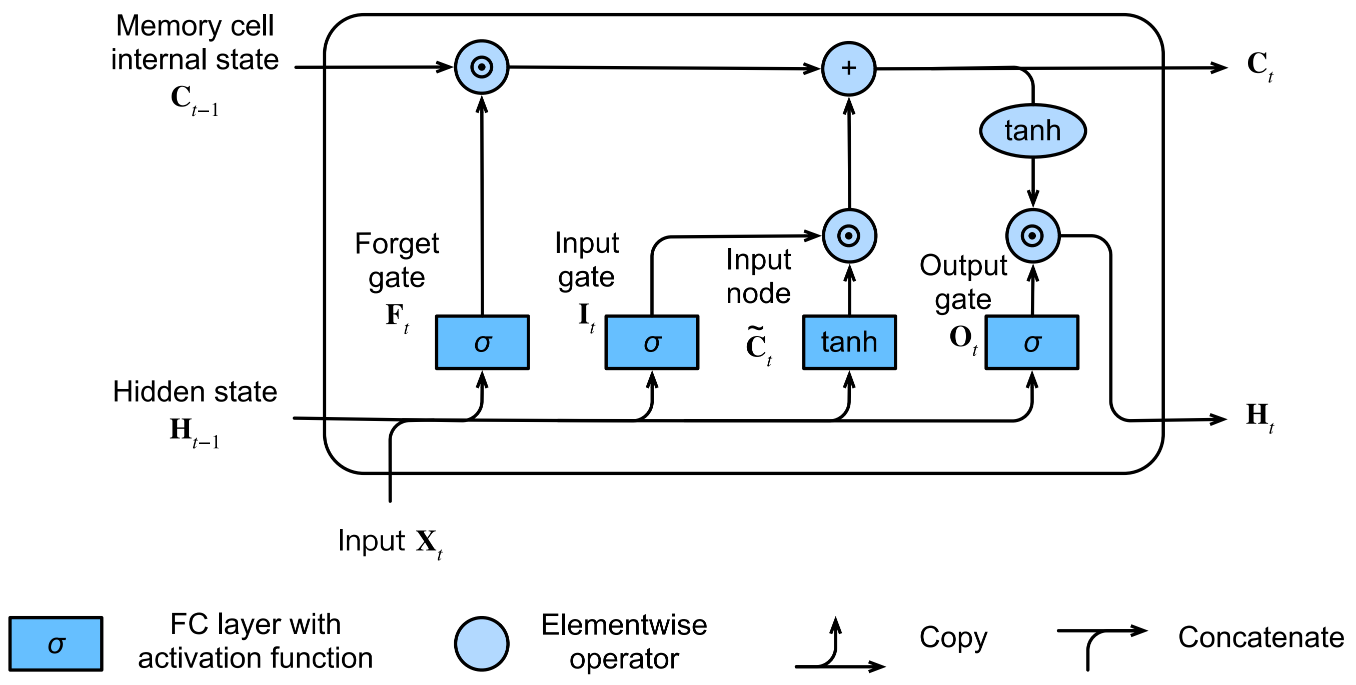
- Shortly after the first RNNs were trained using backpropagation, the problems of learning long-term dependencies (owing to vanishing and exploding gradients) became salient.
- One of the first and most successful techniques for addressing vanishing gradients came in the form of the long short-term memory (LSTM) model due to Hochreiter and Schmidhuber (1997).

[Hochreiter & Schmidhuber, 1997] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735–1780.

- The term “long short-term memory” comes from the following intuition.
- Simple recurrent neural networks have long-term memory in the form of weights. The weights change slowly during training, encoding general knowledge about the data.
- They also have short-term memory in the form of ephemeral activations, which pass from each node to successive nodes.
- The LSTM model introduces an intermediate type of storage via the memory cell. A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with the novel inclusion of multiplicative nodes.

Gated Memory Cell

- Each memory cell is equipped with an internal state and a number of multiplicative gates that determine whether
  - (i) a given input should impact the internal state (the input gate),
  - (ii) the internal state should be flushed to 0 (the forget gate), and
  - (iii) the internal state of a given neuron should be allowed to impact the cell’s output (the output gate).



$$\begin{aligned}
\mathbf{I}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i\right) \in (0,1) \\
\mathbf{F}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right) \in (0,1) \\
\mathbf{O}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o\right) \in (0,1) \\
\tilde{\mathbf{C}}_t &= \tanh\left(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c\right) \in (-1,1)
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\mathbf{C}_t &= \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \\
\mathbf{H}_t &= \mathbf{O}_t \odot \tanh\left(\mathbf{C}_t\right)
\end{aligned}$$

$\odot$ is Hadamard (elementwise) product operator

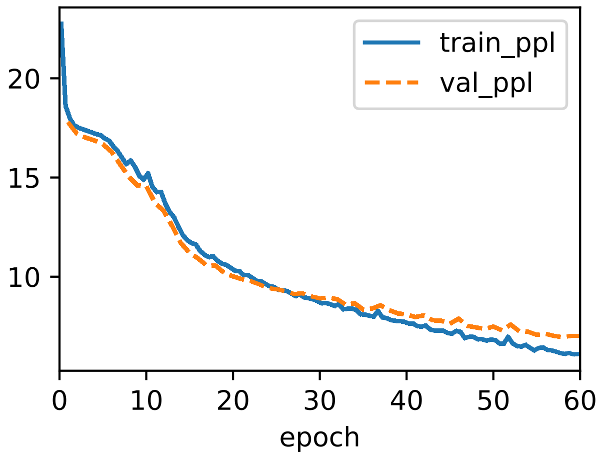
```python
import torch
from torch import nn
from d2l import torch as d2l
import collections
import math
from torch.nn import functional as F

print(torch.__version__)
print(torch.cuda.is_available())
device = "cuda" if torch.cuda.is_available() else "cpu"
print(torch.cuda.get_arch_list(), device)
```

```
1.12.0
True
['sm_37', 'sm_50', 'sm_60', 'sm_61', 'sm_70', 'sm_75', 'sm_80', 'sm_86', 'compute_37'] cuda
```

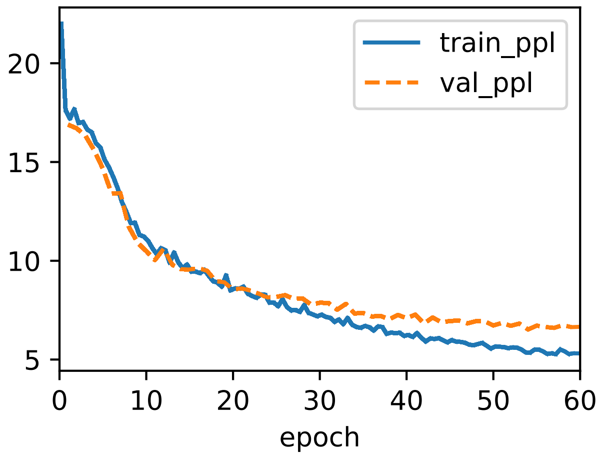
```python
class LSTMScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        init_weight = lambda *shape: nn.Parameter(torch.randn(*shape) * sigma)
        triple = lambda: (init_weight(num_inputs, num_hiddens),
                          init_weight(num_hiddens, num_hiddens),
                          nn.Parameter(torch.zeros(num_hiddens)))
        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

```python
@d2l.add_to_class(LSTMScratch)
def forward(self, inputs, H_C=None):
    if H_C is None:
        # Initial state with shape: (batch_size, num_hiddens)
        H = torch.zeros((inputs.shape[1], self.num_hiddens),
                      device=inputs.device)
        C = torch.zeros((inputs.shape[1], self.num_hiddens),
                      device=inputs.device)
    else:
        H, C = H_C
    outputs = []
    for X in inputs:
        I = torch.sigmoid(torch.matmul(X, self.W_xi) +
                        torch.matmul(H, self.W_hi) + self.b_i)
        F = torch.sigmoid(torch.matmul(X, self.W_xf) +
                        torch.matmul(H, self.W_hf) + self.b_f)
        O = torch.sigmoid(torch.matmul(X, self.W_xo) +
                        torch.matmul(H, self.W_ho) + self.b_o)
        C_tilde = torch.tanh(torch.matmul(X, self.W_xc) +
                           torch.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * torch.tanh(C)
        outputs.append(H)
    return outputs, (H, C)
```

```python
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
lstm = LSTMScratch(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLMScratch(lstm, vocab_size=len(data.vocab), lr=4)
trainer = d2l.Trainer(max_epochs=60, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



```python
class LSTM(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.LSTM(num_inputs, num_hiddens)

    def forward(self, inputs, H_C=None):
        return self.rnn(inputs, H_C)

lstm = LSTM(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLM(lstm, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```



```python
model.predict('time traveller', 20, data.vocab, d2l.try_gpu())
```

```
'time traveller and the time and an'
```

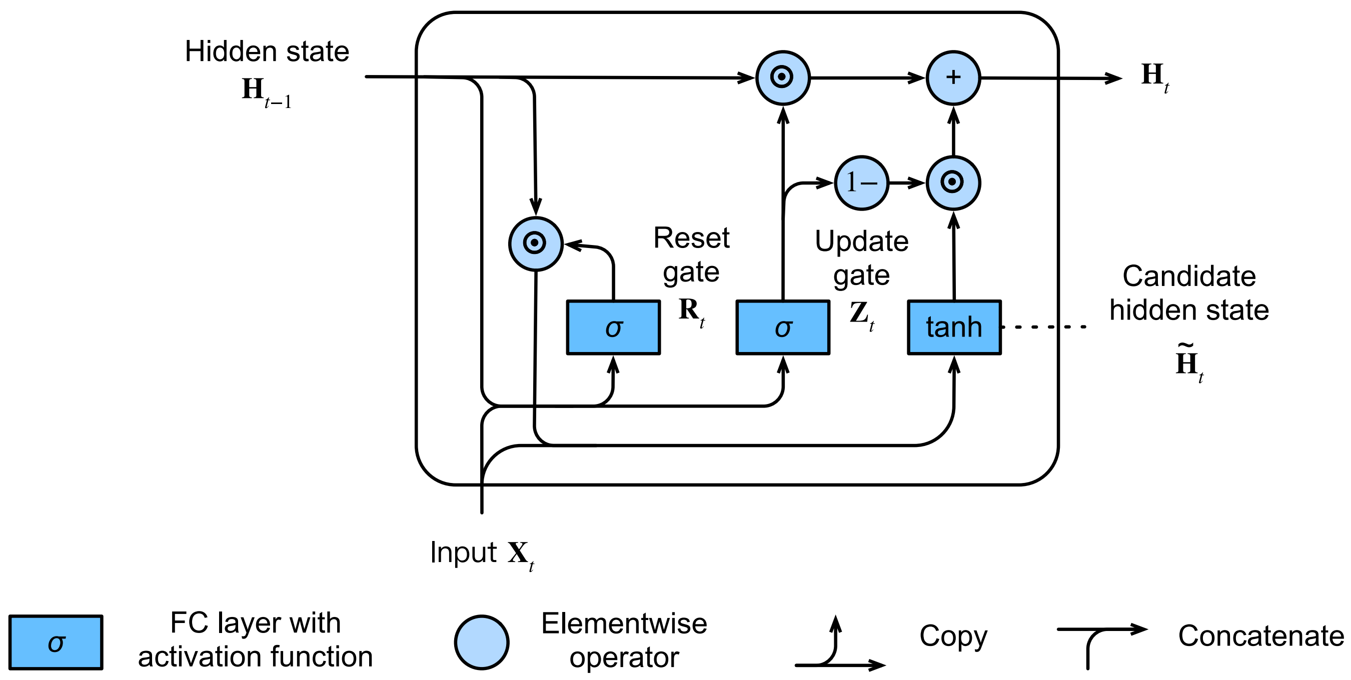
Gated Recurrent Units (GRU)

- RNNs and particularly the LSTM architecture rapidly gained popularity during the 2010s, but researchers want to speed up.
- The gated recurrent unit (GRU) offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute (Chung et al., 2014).

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

Reset Gate and Update Gate

- Here, the LSTM's three gates are replaced by two: the reset gate and the update gate.
- As with LSTMs, these gates are given sigmoid activations, forcing their values to lie in the interval (0, 1).
- Intuitively, the reset gate controls how much of the previous state we might still want to remember. Likewise, an update gate would allow us to control how much of the new state is just a copy of the old state.

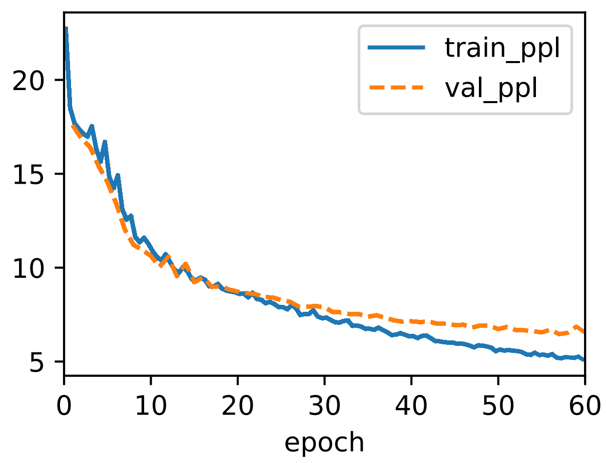


$$
\begin{aligned}
\mathbf{R}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)\\
\mathbf{Z}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)\\
\tilde{\mathbf{H}}_t &= \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h)\\
\mathbf{H}_t &= \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1-\mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t
\end{aligned}
$$

```python
class GRUScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        init_weight = lambda *shape: nn.Parameter(torch.randn(*shape) * sigma)
        triple = lambda: (init_weight(num_inputs, num_hiddens),
                          init_weight(num_hiddens, num_hiddens),
                          nn.Parameter(torch.zeros(num_hiddens)))
        self.W_xz, self.W_hz, self.b_z = triple()  # Update gate
        self.W_xr, self.W_hr, self.b_r = triple()  # Reset gate
        self.W_xh, self.W_hh, self.b_h = triple()  # Candidate hidden state
```

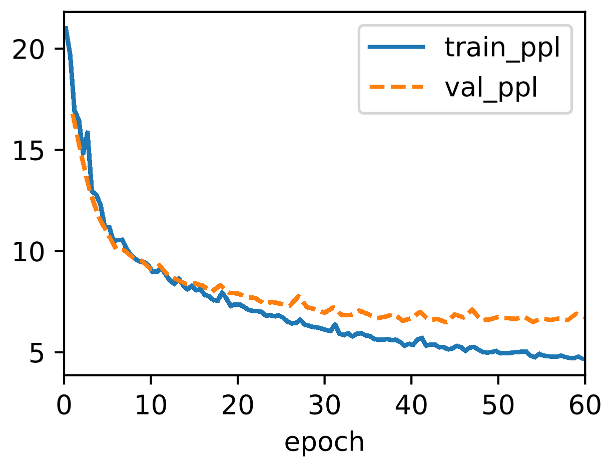
```python
@d2l.add_to_class(GRUScratch)
def forward(self, inputs, H=None):
    if H is None:
        # Initial state with shape: (batch_size, num_hiddens)
        H = torch.zeros((inputs.shape[1], self.num_hiddens),
                      device=inputs.device)
    outputs = []
    for X in inputs:
        Z = torch.sigmoid(torch.matmul(X, self.W_xz) +
                        torch.matmul(H, self.W_hz) + self.b_z)
        R = torch.sigmoid(torch.matmul(X, self.W_xr) +
                        torch.matmul(H, self.W_hr) + self.b_r)
        H_tilde = torch.tanh(torch.matmul(X, self.W_xh) +
                           torch.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        outputs.append(H)
    return outputs, H
```

```python
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
gru = GRUScratch(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLMScratch(gru, vocab_size=len(data.vocab), lr=4)
trainer = d2l.Trainer(max_epochs=60, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



```python
class GRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.GRU(num_inputs, num_hiddens)
```

```python
gru = GRU(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```

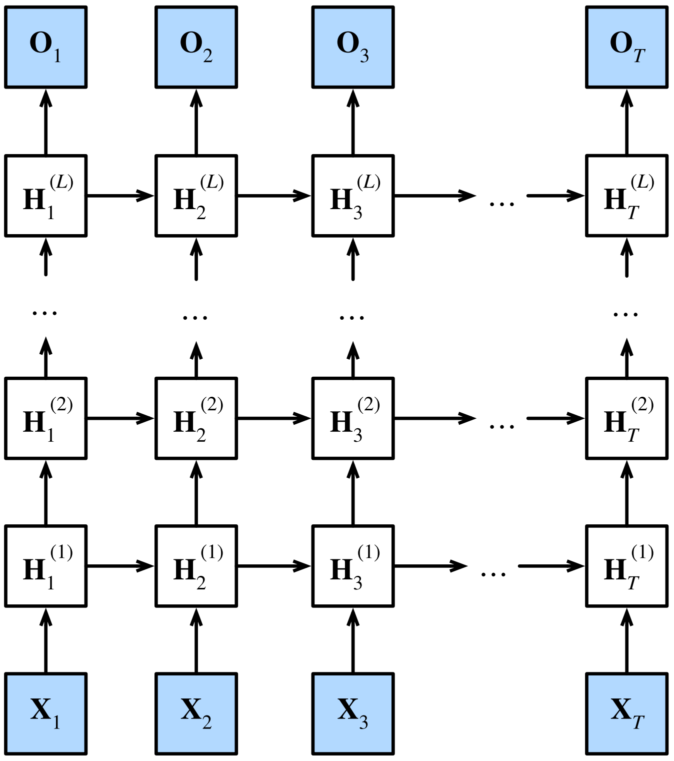


```python
model.predict('time traveller', 20, data.vocab, d2l.try_gpu())
```

```
'time traveller so i some the prect'
```

Deep Recurrent Neural Networks

- The standard method for building this sort of deep RNN is strikingly simple: we stack the RNNs on top of each other. Given a sequence of length T, the first RNN produces a sequence of outputs, also of length T.
- These, in turn, constitute the inputs to the next RNN layer. In this short section, we illustrate this design pattern and present a simple example for how to code up such stacked RNNs.
- Below, in Fig., we illustrate a deep RNN with L hidden layers. Each hidden state operates on a sequential input and produces a sequential output. Moreover, any RNN cell (white box) at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.

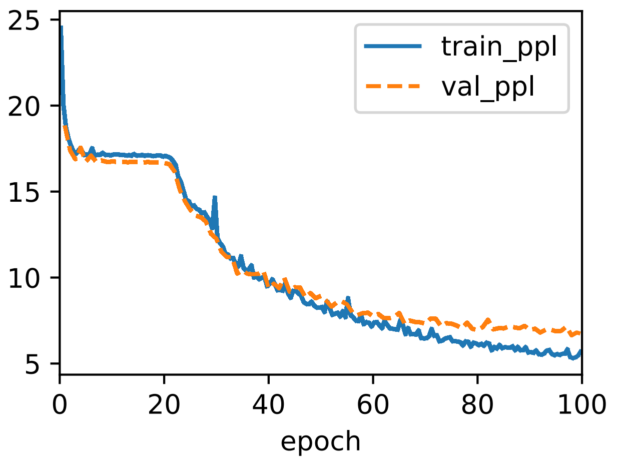


$$\mathbf{H}_t^{(l)} = \phi_l\left(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}\right)$$

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q$$

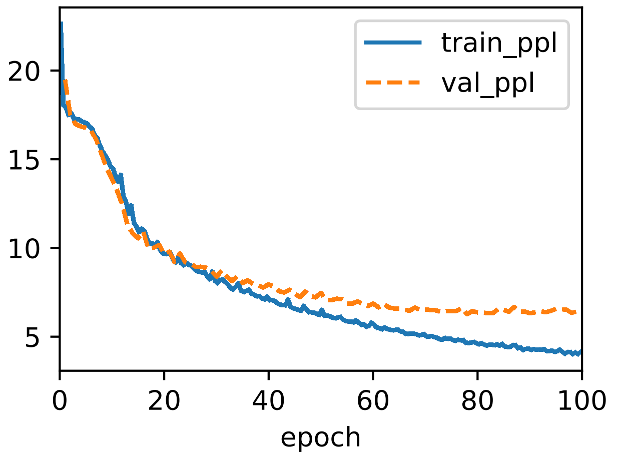
```python
class StackedRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, num_layers, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnns = nn.Sequential(*[d2l.RNNScratch(
            num_inputs if i==0 else num_hiddens, num_hiddens, sigma)
                                    for i in range(num_layers)])

@d2l.add_to_class(StackedRNNScratch)
def forward(self, inputs, Hs=None):
    outputs = inputs
    if Hs is None: Hs = [None] * self.num_layers
    for i in range(self.num_layers):
        outputs, Hs[i] = self.rnns[i](./8-3/outputs, Hs[i])
        outputs = torch.stack(outputs, 0)
    return outputs, Hs
```

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
rnn_block = StackedRNNScratch(num_inputs=len(data.vocab),
                              num_hiddens=32, num_layers=2)
model = d2l.RNNLMScratch(rnn_block, vocab_size=len(data.vocab), lr=2)
trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



```
class GRU(d2l.RNN):  #@save
    """The multi-layer GRU model."""
    def __init__(self, num_inputs, num_hiddens, num_layers, dropout=0):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.GRU(num_inputs, num_hiddens, num_layers,
                          dropout=dropout)
```

```
gru = GRU(num_inputs=len(data.vocab), num_hiddens=32, num_layers=2)
model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
trainer.fit(model, data)
```



```
model.predict('time traveller', 20, data.vocab, d2l.try_gpu())
```

```
'time traveller and the pare and th'
```

Bidirectional Recurrent Neural Networks

- So far, our working example of a sequence learning task has been language modeling, where we aim to predict the next token given all previous tokens in a sequence.
- However, noting that depending on what comes after the blank, the likely value of the missing token changes dramatically:

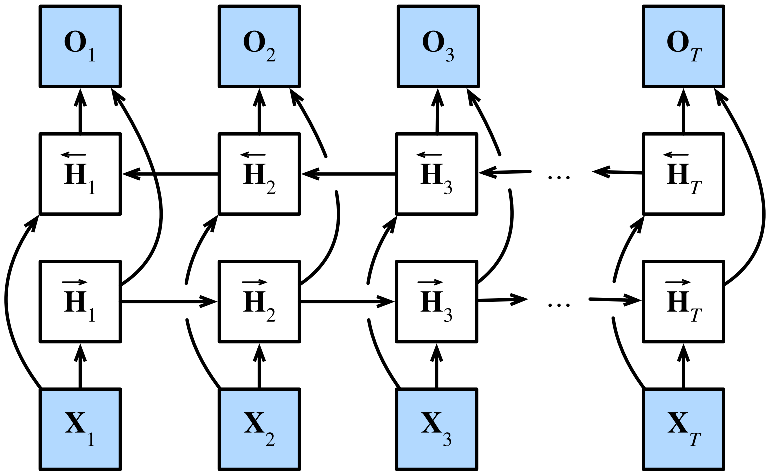
I am ___.

I am ___ hungry.

I am ___ hungry, and I can eat half a pig.

- Fortunately, a simple technique transforms any unidirectional RNN into a bidirectional RNN (Schuster and Paliwal, 1997). We simply implement two unidirectional RNN layers chained together in opposite directions and acting on the same input.

[Schuster and Paliwal, 1997] Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681.



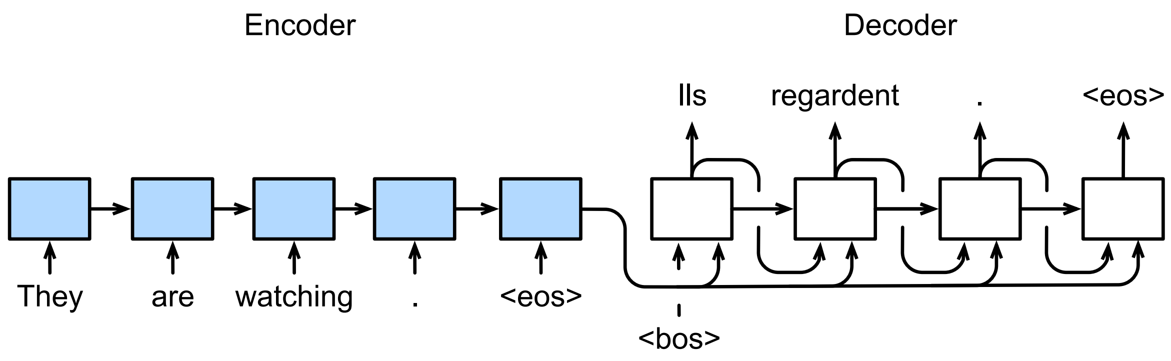
$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\overleftarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_3$
$\overleftarrow{\mathbf{H}}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
$\overrightarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

$$
\begin{aligned}
\overrightarrow{\mathbf{H}}_t &= \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}\right)\\
\overleftarrow{\mathbf{H}}_t &= \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}\right)\\
\mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q
\end{aligned}
$$

```python
class BiRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.f_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.b_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.num_hiddens *= 2  # The output dimension will be doubled

@d2l.add_to_class(BiRNNScratch)
def forward(self, inputs, Hs=None):
    f_H, b_H = Hs if Hs is not None else (None, None)
    f_outputs, f_H = self.f_rnn(inputs, f_H)
    b_outputs, b_H = self.b_rnn(reversed(inputs), b_H)
    outputs = [torch.cat((f, b), -1) for f, b in zip(
        f_outputs, reversed(b_outputs))]
    return outputs, (f_H, b_H)
```

Encoder-Decoder Seq2Seq for Machine Translation

- The encoder RNN will take a variable-length sequence as input and transform it into a fixed-shape hidden state.
- Then to generate the output sequence, one token at a time, the decoder model, consisting of a separate RNN, will predict each successive target token given both the input sequence and the preceding tokens in the output.
- During training, the decoder will typically be conditioned upon the preceding tokens in the official "ground-truth" label. However, at test time, we will want to condition each output of the decoder on the tokens already predicted.



Teacher Forcing

- The special <bos> token and the original target sequence, excluding the final token, are concatenated as input to the decoder, while the decoder output (labels for training) is the original target sequence, shifted by one token:
- "<bos>", "Ils", "regardent", "." $\rightarrow$ "Ils", "regardent", ".", "<eos>"

Encoder

- We can use a function f to express the transformation of the RNN's recurrent layer.
- In general, the encoder transforms the hidden states at all time steps into a context variable through a customized function q.

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$
$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T)$$

```python
def init_seq2seq(module):  #@save
    """Initialize weights for Seq2Seq."""
    if type(module) == nn.Linear:
         nn.init.xavier_uniform_(module.weight)
    if type(module) == nn.GRU:
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[param])

class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size, num_hiddens, num_layers, dropout)
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(X.t().type(torch.int64))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = torch.zeros((batch_size, num_steps))
enc_outputs, enc_state = encoder(X)
print(enc_outputs.shape)
print(num_steps, batch_size, num_hiddens)
print("-"*40)
print(enc_state.shape)
print(num_steps, batch_size, num_hiddens)
```

```
torch.Size([9, 4, 16])
9 4 16
----------------------------------------
torch.Size([2, 4, 16])
9 4 16
```

Decoder

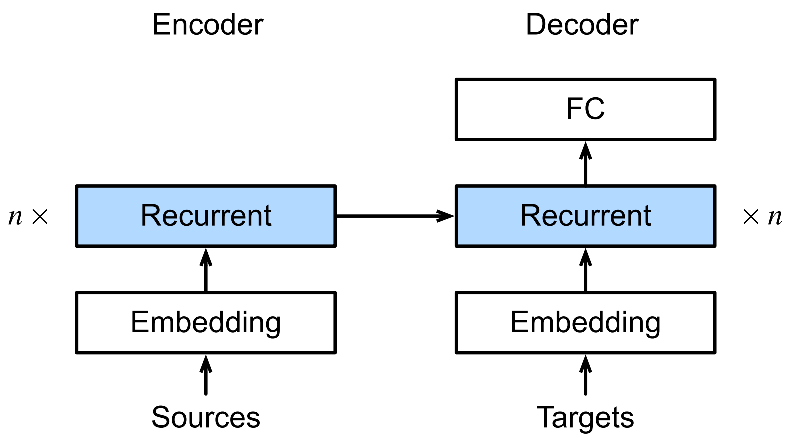
- the decoder assigns a predicted probability to each possible token occurring at step $y_{t'+1}$ conditioned upon the previous tokens in the target $y_1, \ldots, y_{t'}$ and the context variable $\mathbf{c}$, i.e., $y_{t'+1} = P\left(y_{t'+1} \mid y_1, \ldots, y_{t'}, \mathbf{c}\right)$.
- To predict the subsequent token $t'+1$ in the target sequence, the RNN decoder takes the previous step's target token $y_{t'}$, the hidden RNN state from the previous time step $\mathbf{s}_{t'-1}$, and the context variable $\mathbf{c}$ as its input, and transforms them into the hidden state $\mathbf{s}_{t'}$ at the current time step.

$$\mathbf{s}_{t'} = g\left(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}\right)$$

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size+num_hiddens, num_hiddens,
                           num_layers, dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs
    def forward(self, X, state):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(X.t().type(torch.int32))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = context.repeat(embs.shape[0], 1, 1)
        # Concat at the feature dimension
        embs_and_context = torch.cat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        outputs = self.dense(outputs).swapaxes(0, 1)
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

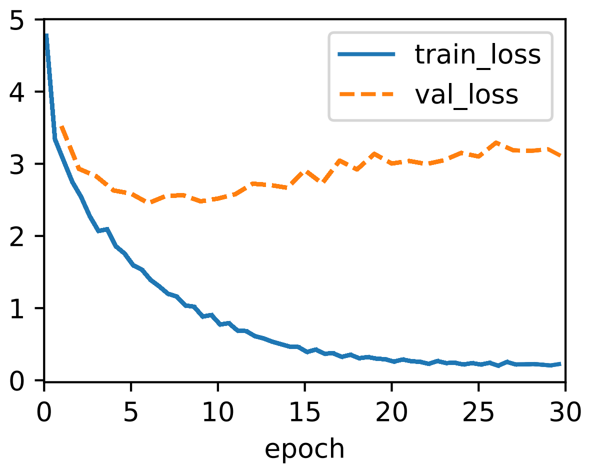
```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
state = decoder.init_state(encoder(X))
dec_outputs, state = decoder(X, state)
print(dec_outputs.shape)
print(batch_size, num_steps, vocab_size)
print("-"*40)
print(state[1].shape)
print(num_layers, batch_size, num_hiddens)
```

```
torch.Size([4, 9, 10])
4 9 10
----------------------------------------
torch.Size([2, 4, 16])
2 4 16
```



```
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder-decoder for sequence to sequence learning."""
    def __init__(self, encoder, decoder, tgt_pad, lr):
        super().__init__(encoder, decoder)
        self.save_hyperparameters()
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        return torch.optim.Adam(self.parameters(), lr=self.lr)

@d2l.add_to_class(Seq2Seq)
def loss(self, Y_hat, Y):
    l = super(Seq2Seq, self).loss(Y_hat, Y, averaged=False)
    mask = (Y.reshape(-1) != self.tgt_pad).type(torch.float32)
    return (l * mask).sum() / mask.sum()
```

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
encoder = Seq2SeqEncoder(
    len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                lr=0.005)
trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



```
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
    batch = [a.to(device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    enc_all_outputs = self.encoder(src, src_valid_len)
    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [tgt[:, 0].unsqueeze(1), ], []
```

```
        for _ in range(num_steps):
            Y, dec_state = self.decoder(outputs[-1], dec_state)
            outputs.append(Y.argmax(2))
            # Save attention weights (to be covered later)
            if save_attention_weights:
                attention_weights.append(self.decoder.attention_weights)
        return torch.cat(outputs[1:], 1), attention_weights
```

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
preds, _ = model.predict_step(
    data.build(engs, fras), d2l.try_gpu(), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['il', 'est', 'mouillé', '.'], bleu,0.658
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```